# ­­­­­Interpretazione del testo e scelte progettuali

L’obiettivo di questo documento è quello di fornire un quadro generale della interpretazione e delle scelte progettuali che sono state prese per il progetto e a tal proposito saranno chiariti i seguenti aspetti:

1. Dalle specifiche non è definito se i nodi degli alberi di espressioni debbano essere univocamente identificabili dal loro tag. Si assume che sia possibile avere più nodi con lo stesso tag e le conseguenze di questa scelta saranno chiarite nei punti successivi.
2. I tipi esprimibili sono stati estesi con gli alberi di espressioni in quanto, come specificato dal testo, le nuove operazioni Select e ApplyOver devono restituire un albero di espressioni opportunamente valutato.
3. Il type checker del linguaggio è stato esteso in modo da poter riconoscere gli alberi di espressioni.
4. E’ stato aggiunto il seguente pattern alle funzioni primitive in modo da eliminare i warning prodotti dal compilatore a causa dei pattern matching non esaustivi: \_ -> **failwith**("Type error - typecheck failed")
5. Di seguito le estensioni fatte alla funzione di valutazione delle espressioni del linguaggio “eval”:
   * 1. **Select(i,et):** affinché eval possa valutare correttamente questa espressione, occorre che “i” rappresenti un identificatore (ide), mentre “et” un albero di espressioni (ETree). Se le premesse sono soddisfatte eval cerca, con una visita anticipata dell’albero (sequenza di visita ricorsiva: nodo, sottoalbero sinistro, sottoalbero destro), il primo nodo con tag uguale a "i". Se un nodo con tag uguale a "i" viene individuato, viene restituito l’albero radicato in esso, altrimenti viene restituito un albero vuoto. L’ipotesi fatta nel punto 1 ha la seguente conseguenza:

All’interno di un albero possono coesistere nodi diversi con lo stesso tag, perciò il tipo di visita che viene scelto (in questo caso anticipata) determina quale nodo verrà selezionato in caso di più nodi con lo stesso tag. Di seguito alcuni esempi di interpretazione di Select con tipi di visita differenti:

|  |  |  |  |
| --- | --- | --- | --- |
| tree | Tipo visita | Comando | Risultato |
| b| 1  b | 2  b | 3 | Anticipata (**utilizzata da eval**): nodo, albero sinistro, albero destro | eval Select(“b”,tree) env0;; | Tree(Node ("b", Int 1,  Tree (Node ("b", Int 2, Tree Empty, Tree Empty)),  Tree (Node ("b", Int 3, Tree Empty, Tree Empty)))) |
| Simmetrica (utilizzata da eval): albero sinistro, nodo, albero destro | eval Select(“b”,tree) env0;; | Tree(Node ("b", Int 2,Tree Empty, Tree Empty)) |
| Posticipata (utilizzata da eval): albero sinistro, albero destro, nodo | eval Select(“b”,tree) env0;; | Tree(Node ("b", Int 2,Tree Empty, Tree Empty)) |

* + 1. **ApplyOver(tags,f,et):** affinché eval possa valutare correttamente questa espressione occorre che tags rappresenti una lista di identificatori (ide list), “f” una funzione anonima (Fun) e “et” un albero di espressioni (ETree). Se le premesse sono soddisfatte eval applica la funzione f su tutti i nodi dell’albero “et” aventi un tag presente nella list “tags”. In questo caso la scelta fatta nel punto 1 è ininfluente, perché in ogni caso tutto l’albero viene visitato e l’ordine con cui viene fatto non ha importanza.

# Compilazione ed esecuzione

Il file “evalFunEnvFull.ml” contiene il codice del linguaggio visto durante il corso, mentre il file “extended\_evalFunEnvFull.ml” contiene il codice esteso per poter gestire gli alberi di espressioni, più una batteria di test che ha lo scopo di valutare che l’estensione sia stata effettuata correttamente.

|  |  |
| --- | --- |
| Comando shell | Descrizione |
| ocamlbuild extended\_evalFunEnvFull.native | Compila il programma ocaml “extended\_evalFunEnvFull.ml”. Per eseguire il programma occorre eseguire il file generato dalla compilazione: “extended\_evalFunEnvFull.native” |
| ocamlbuild extended\_evalFunEnvFull.native -- | Compila ed esegue il programma ocaml “extended\_evalFunEnvFull.ml”. |